

***Report on***

**“C++ Compiler Using Lex and Yacc”**

***Compiler Design Laboratory***

**Bachelor of Technology in**

**Computer Science & Engineering**

***Submitted by:***

|  |  |
| --- | --- |
| **Vansh angaria Krishnamurthi** | **122cs0010 122cs0067** |

*Under the guidance of*

**Dr K Nagaraju** Assistant Professor IIITDM,Kurnool

**TABLE OF CONTENTS**

|  |  |  |
| --- | --- | --- |
| **Chapter No.** | **Title** | **Page No.** |
| **1.** | **INTRODUCTION (Mini-Compiler is built for which language. Provide sample input and output of your project)** | **04** |
| **2.** | **ARCHITECTURE OF LANGUAGE:**   * **What all have you handled in terms of syntax and semantics for the chosen language.** | **05** |
| **3.** | **LITERATURE SURVEY (if any paper referred or link used)** | **06** |
| **4.** | **CONTEXT FREE GRAMMAR (which you used to implement your project)** | **07-11** |
| **5.** | **DESIGN STRATEGY (used to implement the following)**   * **SYMBOL TABLE CREATION** * **ABSTRACT SYNTAX TREE** * **INTERMEDIATE CODE GENERATION** * **CODE OPTIMIZATION** * **ERROR HANDLING *-* strategies and solutions used in your Mini-Compiler implementation (in its scanner, parser, semantic analyzer, and code generator).** * **TARGET CODE GENERATION** | **12-13** |
| **6.** | **IMPLEMENTATION DETAILS (TOOL AND DATA**  **STRUCTURES USED in order to implement the following):**   * **SYMBOL TABLE CREATION** * **ABSTRACT SYNTAX TREE (internal representation)** * **INTERMEDIATE CODE GENERATION** * **CODE OPTIMIZATION** * **ASSEMBLY CODE GENERATION** * **ERROR HANDLING *-* strategies and solutions used in your Mini-Compiler implementation (in its scanner, parser, semantic analyzer, and code generator).** * **Provide instructions on how to build and run your program.** | **14-15** |
| **7.** | **RESULTS AND possible shortcomings of your Mini-Compiler** | **16-17** |
| **8.** | **SNAPSHOTS (of different outputs)** | **18-23** |
| **9.** | **CONCLUSIONS** | **24** |
| **10.** | **FURTHER ENHANCEMENTS** |  |
| **REFERENCES/BIBLIOGRAPHY** | |  |

# Introduction

C++ is one of the world's most popular programming languages.C++ can be found in today's operating systems, Graphical User Interfaces, and embedded systems.C++ is an object-oriented programming language which gives a clear structure to programs and allows code to be reused, lowering development costs.C++ is portable and can be used to develop applications that can be adapted to multiple platforms.

The objective of our project was to create a simple C++ compiler which can convert the given C++ file into an executable ARM assembly program.

Any compiler follows certain process involving :

* Lexical Phase
* Syntax Parser
* Semantic Parser
* Intermediate Code Generation
* ICG Optimization
* Assembly Code Generation.

Our project involves performing all the above processes step by step in different phases.

# Architecture of Language

This project deals with building a simple C++ compiler using Flex for the lexical phase, Yacc for abstract syntax tree and intermediate code generation and python for code optimization and assembly code generation.

The C++ compiler built deals with the following conditional and iterative programming constructs:

1. Conditional Constructs:
   1. if
   2. if-else
2. Iterative Constructs:
   1. for
   2. while

A C++ input file is given as an input to the compiler as follows:

# Literature Survey

The following sources were referenced for this project:

<https://www.javatpoint.com/lex>

<https://www.geeksforgeeks.org/introduction-to-yacc/>

<https://ruslanspivak.com/lsbasi-part7/>

<https://compileroptimizations.com/category/constant_propagation.htm>

<https://compileroptimizations.com/category/constant_folding.htm>

<https://compileroptimizations.com/category/dead_code_elimination.htm>

<https://www.javatpoint.com/code-generation>

[https://web.cs.ucdavis.edu/~pandey/Teaching/ECS142/Lects/final.code](https://web.cs.ucdavis.edu/~pandey/Teaching/ECS142/Lects/final.codegen.pdf) [gen.pdf](https://web.cs.ucdavis.edu/~pandey/Teaching/ECS142/Lects/final.codegen.pdf)

<https://compileroptimizations.com/category/constant_propagation.htm>

# Context-Free Grammar

S : program

;

program

: HASH INCLUDE '<' libraries '>' program

| HASH INCLUDE HEADER\_LITERAL program

| NSPACE translation\_unit

; NSPACE

: NAMESPACE ';'

translation\_unit

: ext\_dec

| translation\_unit ext\_dec

;

ext\_dec

: declaration

| function\_definition

;

libraries

: CSTDIO

| STDLIB

| MATH

| STRING

| TIME

| IOSTREAM

| CONIO

;

compound\_statement

: '{' '}'

| '{' block\_item\_list '}'

;

block\_item\_list

: block\_item

| block\_item\_list block\_item

;

block\_item

: declaration

| statement

| RETURN expression\_statement

| function\_call ';'

| printstat ';'

;

printstat

: PRINT '(' STRING\_LITERAL ')'

| PRINT '(' STRING\_LITERAL ',' expression ')'

| COUT '<''<' STRING\_LITERAL

;

declaration

: type\_specifier init\_declarator\_list ';'

;

statement

: compound\_statement

| expression\_statement

| iteration\_statement

| conditional\_statement

;

conditional\_statement

: IF '('

conditional\_expression ')' statement conditional\_statement;

| ELSE

|

;

iteration\_statement

:FOR '(' expression\_statement expression\_statement

expr ')' statement

| WHILE '(' conditional\_expression ')'

;

expr

: IDENTIFIER INC\_OP

| IDENTIFIER DEC\_OP

| INC\_OP IDENTIFIER

| DEC\_OP IDENTIFIER

| IDENTIFIER ADD\_ASSIGN INTEGER\_LITERAL

| IDENTIFIER SUB\_ASSIGN INTEGER\_LITERAL

| IDENTIFIER MUL\_ASSIGN INTEGER\_LITERAL

| IDENTIFIER DIV\_ASSIGN INTEGER\_LITERAL

| IDENTIFIER '=' IDENTIFIER '+' INTEGER\_LITERAL

| IDENTIFIER '=' IDENTIFIER '-' INTEGER\_LITERAL

| IDENTIFIER '=' IDENTIFIER '\*' INTEGER\_LITERAL

| IDENTIFIER '=' IDENTIFIER '/' INTEGER\_LITERAL

;

type\_specifier

: VOID

| CHAR

| INT

| FLOAT

;

init\_declarator\_list

: init\_declarator

| init\_declarator\_list ',' init\_declarator

;

init\_declarator

: IDENTIFIER '=' assignment\_expression

| IDENTIFIER

;

assignment\_expression

: conditional\_expression

| unary\_expression assignment\_operator assignment\_expression

;

assignment\_operator

: '='

| ADD\_ASSIGN

| SUB\_ASSIGN

| MUL\_ASSIGN

| DIV\_ASSIGN

| MOD\_ASSIGN ;

conditional\_expression

: equality\_expression

| equality\_expression '?' expression

':' conditional\_expression

;

expression\_statement

: ';'

| expression ';'

;

expression

: assignment\_expression

| expression ',' assignment\_expression

;

primary\_expression

: IDENTIFIER

| INTEGER\_LITERAL

| FLOAT\_LITERAL

| CHARACTER\_LITERAL

| '(' expression ')'

;

postfix\_expression

: primary\_expression

| postfix\_expression INC\_OP

| postfix\_expression DEC\_OP

;

unary\_expression

: postfix\_expression

| unary\_operator unary\_expression

;

unary\_operator

: '+'

| '-'

| '!'

| '~'

| "INC\_OP"

| "DEC\_OP"

;

equality\_expression

: relational\_expression

| equality\_expression EQ\_OP relational\_expression

| equality\_expression NE\_OP relational\_expression

;

relational\_expression

: additive\_expression

| relational\_expression '<' additive\_expression

| relational\_expression '>' additive\_expression

| relational\_expression LE\_OP additive\_expression

| relational\_expression GE\_OP additive\_expression

;

additive\_expression

: multiplicative\_expression

| additive\_expression '+' multiplicative\_expression

| additive\_expression '-' multiplicative\_expression

;

multiplicative\_expression

: unary\_expression

| multiplicative\_expression '\*' unary\_expression

| multiplicative\_expression '/' unary\_expression

| multiplicative\_expression '%' unary\_expression

;

function\_definition

: type\_specifier declarator compound\_statement

| declarator compound\_statement

;

function\_call

: declarator '(' identifier\_list ')'

| declarator '(' ')'

;

declarator

: IDENTIFIER

| declarator '(' parameter\_list ')'

| declarator '(' identifier\_list ')'

| declarator '(' ')'

;

parameter\_list

: parameter\_declaration

| parameter\_list ',' parameter\_declaration

;

parameter\_declaration

: type\_specifier IDENTIFIER

| type\_specifier

;

identifier\_list

: IDENTIFIER

| identifier\_list ',' IDENTIFIER

;

# Design Strategies

### Symbol Table Generation:

* + In any compiler the symbol table holds details of the identifiers such as the scope,value,identifier name,line number etc. In our implementation of the symbol table,each entry of the symbol table is a structure whose properties include-token type,token value,line # and data type. Scope level is taken care of by a global variable initialized to 0 and is incremented every time the lex encounters an open curly bracket and decrements everytime the lex encounters a closed curly bracket.

### Abstract Syntax Tree:

* + The abstract syntax tree is constructed by first creating a node data structure containing necessary information of the token which will be stored in the AST. Along with the tokens generated by the lex the yacc is used for rule matching and nodes are inserted using the rule part of the grammar in yacc. Implicit data type conversion is done using the symbol table,everytime an assignment operation is encountered based on the value being assigned and the data type stored in the symbol table, the conversion is done.

### Intermediate Code Generation:

* + Intermediate code generation is done in formats in our implementation - 3 address code and quadruple format. For either methods the strategy mainly involves the yacc grammar. Our strategy was to generate the ICG component by component of any statement . For example for generating the ICG for a while loop, first the ICG is generated for the conditional expression inside the while loop parentheses and then generate the ICG individually for the statements inside the while loop.

### Error Handling:

* + As far as our implementation is concerned only basic error handling is taken care of during the syntax validation phase of our compiler. If a rule is not matched, the yacc implicitly makes a call to the yyerror() function where a char pointer is passed as an argument(argument is passed implicitly by the yacc). Based on the error and line number the error is printed.

### Target Code Generation:

* + Target code generated using a python script which reads the icg file and stores it line by line.Line by line the ARM statements are generated and registers are chosen in a round robin fashion(R%13) to ensure that registers are correctly used. For branch conditions, the condition statements are skipped and taken care off when we encounter the *ifFalse* statement. When the ifFalse statement is encountered then the previous statement which will be the conditional statement is converted into an ARM *CMP* statement and the ifFalse is converted to a *B* statement based on the condition.

# Implementation Details

### Symbol Table Creation:

* + Tools used: Flex and GCC
  + Language used: C
  + Data Structures: Custom struct in C to hold symbol table entry details and array of this structure is used to hold symbol table
  + Commands to compile and run the lexical phase:

lex phase1.l gcc lex.yy.c

./a.out < [filename]

### Abstract Syntax Tree:

* + Tools used: Flex,Bison Yacc and GCC
  + Language used: C
  + Data Structures: Internally stored as a binary tree
  + Commands to compile and run the semantic phase:

lex ast.l yacc -d ast.y

gcc lex.yy.c y.tab.c

./a.out < [filename]

### Intermediate Code Generation:

* + Tools used: Flex,Bison Yacc and GCC
  + Language used: C
  + Data Structures: No special data structures are used implicitly.
    - Grammar rules are used to generate ICG component wise
    - Each statement of ICG is written to a text file
  + Commands to compile and run the ICG generator:

lex icg.l yacc -d icg.y

gcc lex.yy.c y.tab.c

./a.out < [filename]

### Code Optimization:

* + Tools used: Python
  + Language used: Python
  + Data Structures: Python list is used to hold list of lines
    - ICG file is read and split into sentences and stored in array
    - After processing,these statements are written back to a text file line by line
  + Commands to run the python optimizer script:

python icg\_opt.py [filename] --print

### Assembly Code Generation:

* + Tools used: Python
  + Language used: Python
  + Data structures:
    - Hash map is used to store a variable list to ensure variables are not loaded from memory again and again.
    - Python list to store ARM generated statements which are written to a **.s** ASM file after processing all ICG statements
  + Commands to run the assembly generator:

python gen.py [filename]

Note: Assembly code will be stored in *filename.s*

### Error Handling:

* + Tools used: Yacc
  + Language used: C
  + Data structures: None
  + Command to run syntax analyzer:

lex lexical.l yacc -d gram.y

gcc lex.yy.c y.tab.c

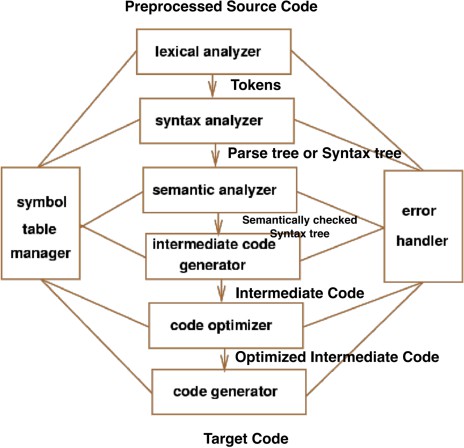
./a.out < [filename]

# Results

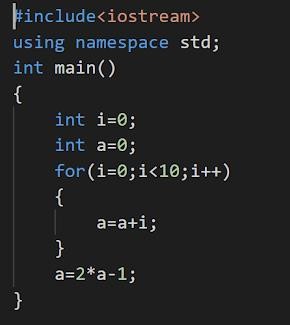
A simple C++ compiler with the following constructs was successfully built:

* for
* while
* if
* if-else

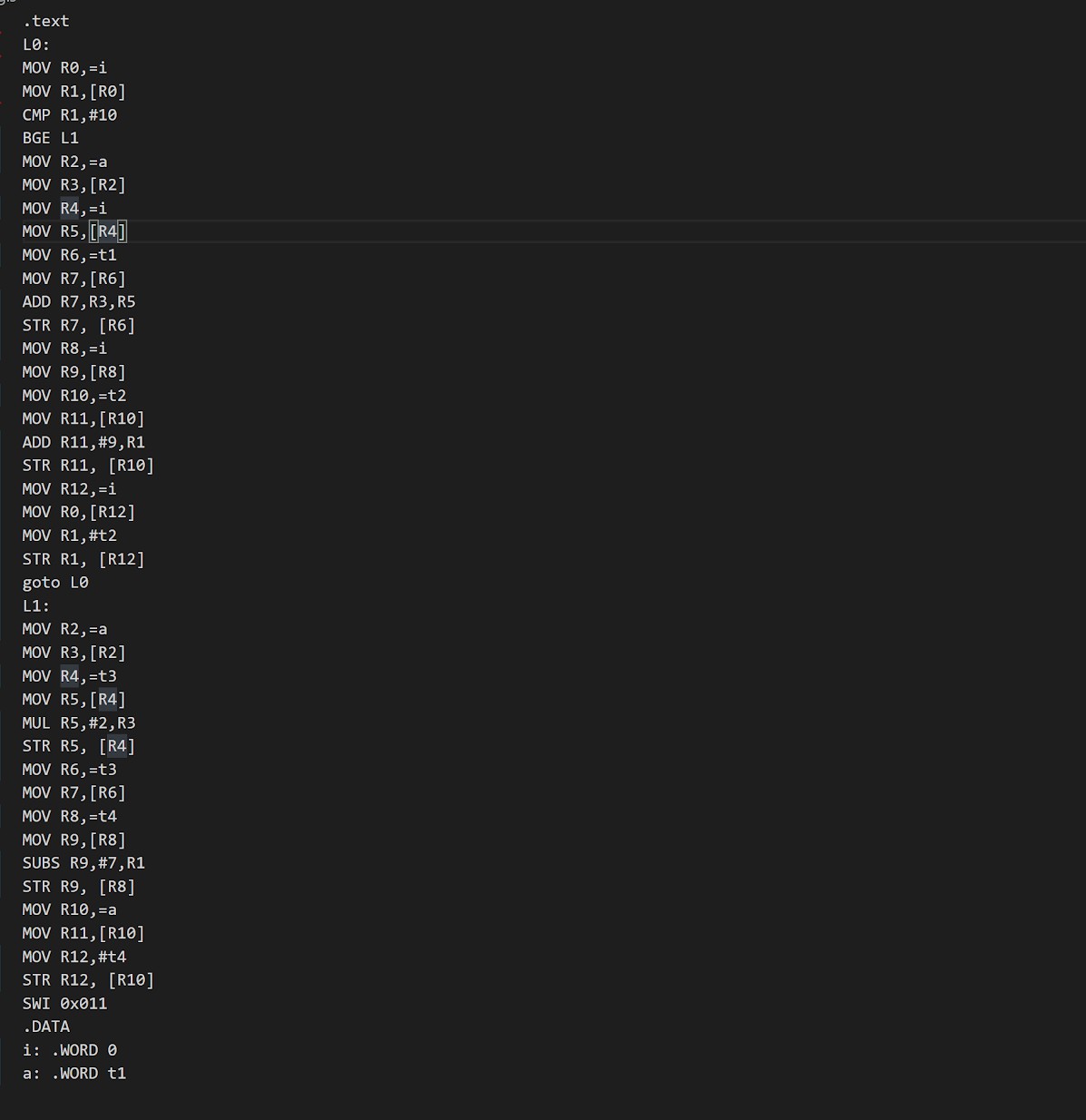
The implemented C++ compiler successfully performs syntax and semantic validation,ICG generation,ICG optimization and assembly code generation as per the below diagram:



An end to end (C++ to ARM) is shown below:



**Input C++ File**



**ARM Assembly Generated for C++ file**

# Draw Drawback:

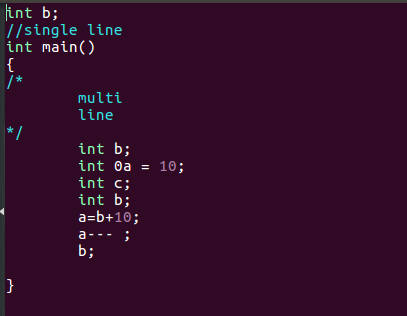
* Assembly code generated use too many registers as it reloads variables again and again

# Future Work:

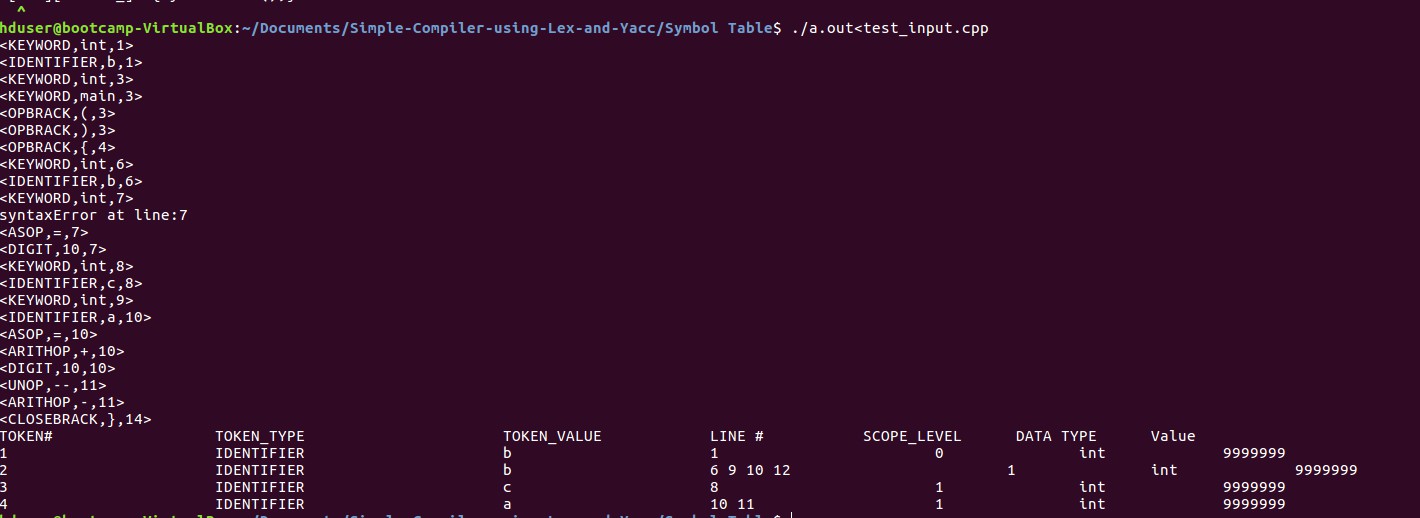
* Include more number of constructs
* Implement arrays
* Implement classes and objects

# Snapshots

## Phase 1: Lexical Phase(Generation of Tokens and Symbol Table)

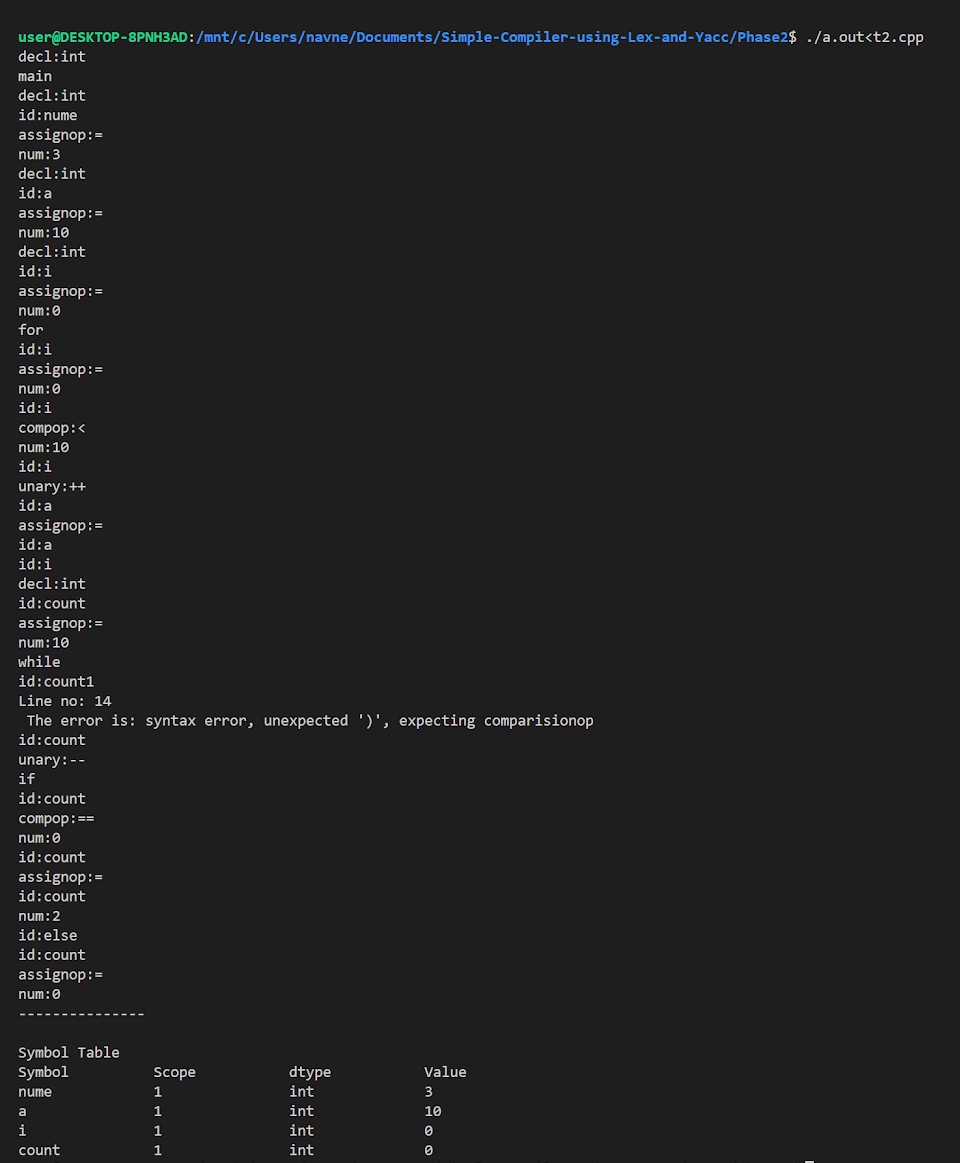
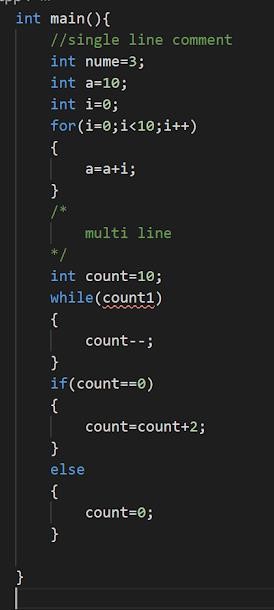
******

**Input File**

****

**Output result- Tokens generated and Symbol Table**

## Phase 2: Syntax Analyzer(Validation of Input With Grammar)

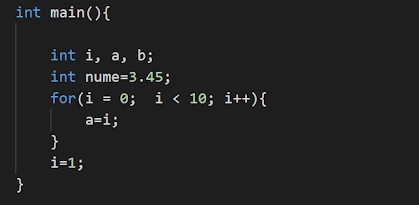
******

**C++ code with error**

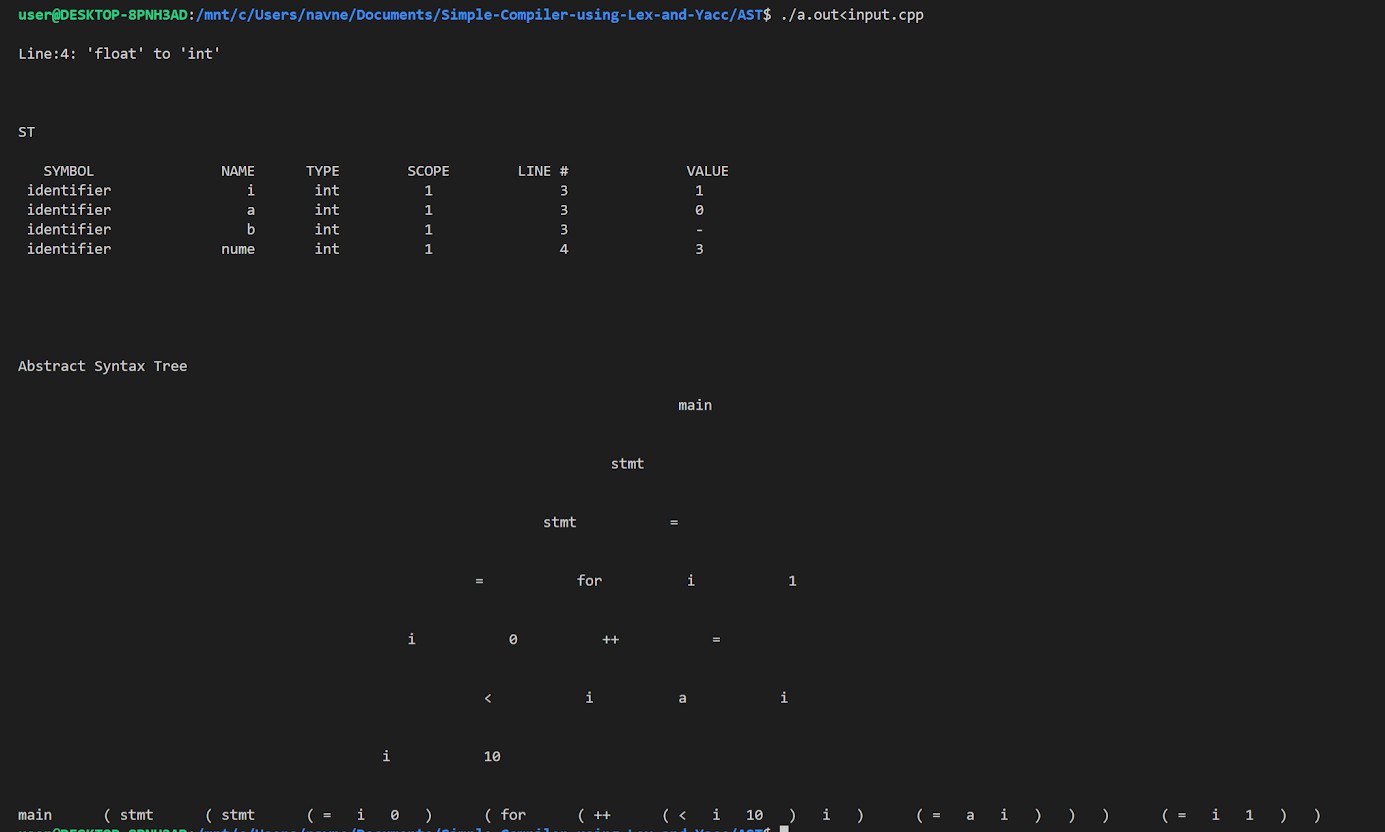
**Syntax Error generated by parser along with tokens and**

**symbol table**

## Phase 3: Semantic Analyzer(Abstract Syntax Tree Generation)

******

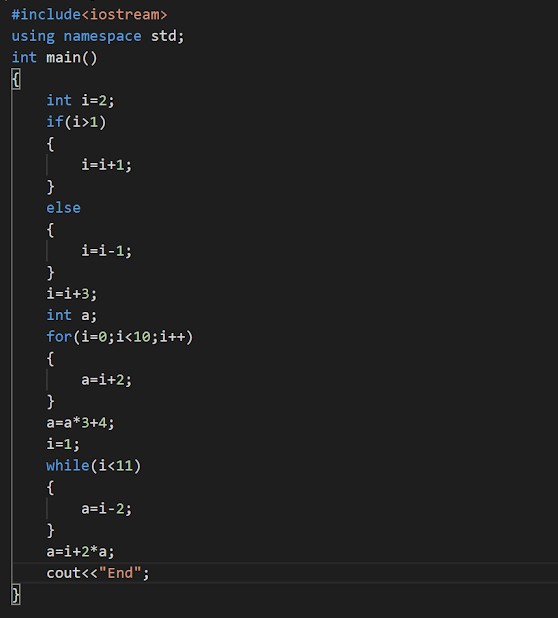
**Input C++ Code**

****

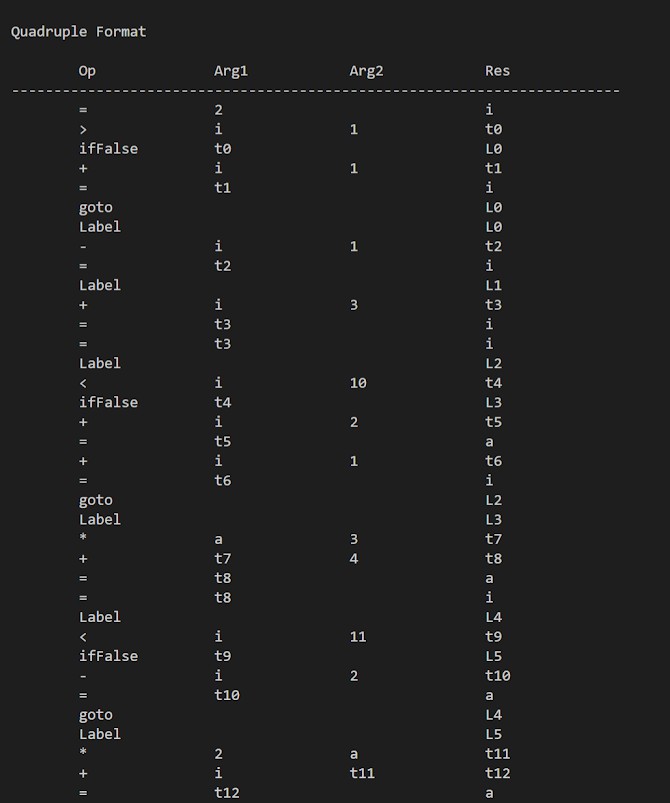
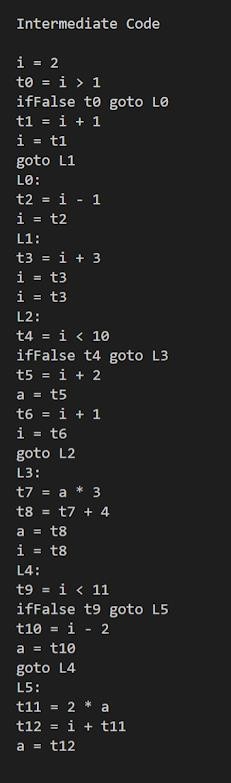
**Implicit float to int conversion Symbol Table**

**AST along with the pre-order traversal**

## Phase 4: Intermediate Code Generation(3 address code and quadruple format)

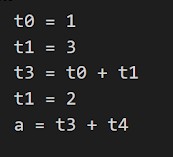
******

**Input C++ File**

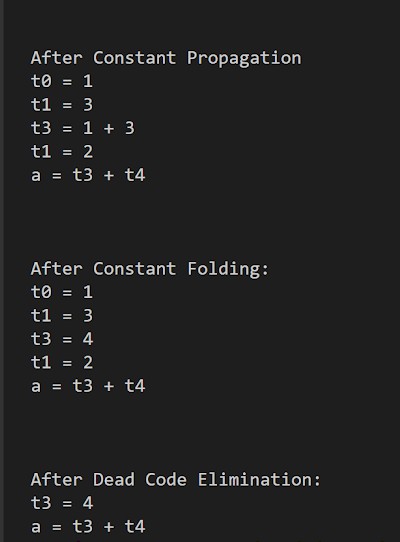
****

**3 address code Quadruple Format ICG ICG**

## Phase 5: ICG Optimization(Constant Propagation,Constant folding and Dead Code Elimination)

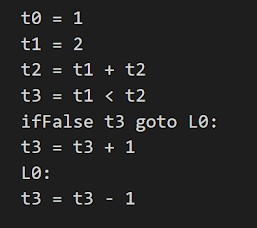
******

**Example ICG Before Optimization**

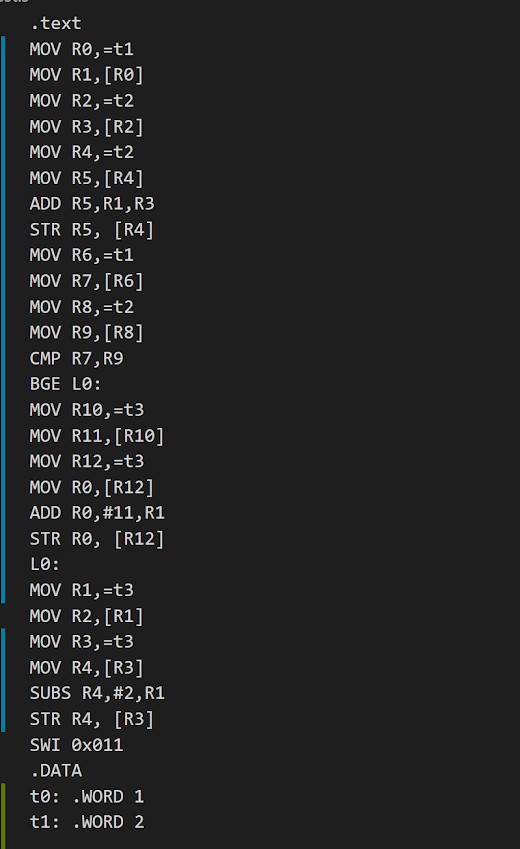
****

**Output of constant propagation,constant folding and dead code elimination**

## Phase 6: Assembly Code Generation(ARM Instruction Set)

******

**ICG code to be converted to target code**

****

**ARM assembly code generated**

# Conclusion:

A simple C++ compiler was successfully built for the previously mentioned constructs as well as all basic C++ programming paradigms.